CHAPTER 11

Partial Differential Equations

Partial differential equations (PDEs) are multivariate differential equations where derivatives of more than one dependent variable occur. That is, the derivatives in the equation are *partial* derivatives. They are generalizations of ordinary differential equations covered in Chapter 9. Conceptually, the difference between ordinary and partial differential equations is small, but the computational techniques required to deal with ODEs and PDEs are very different, and solving PDEs is typically much more computationally demanding. Most techniques for solving PDEs numerically are based on discretizing the problem in each independent variable that occurs in the PDE, thereby recasting the problem into an algebraic form. This usually results in very large-scale linear algebra problems. Two common techniques for recasting PDEs into algebraic form are the finite-difference methods (FDMs), where the derivatives in the problem are approximated with their finite-difference formula, and the finite-element methods (FEMs), where the unknown function is written as a linear combination of simple basis functions that can be differentiated and integrated easily. The unknown function is described by a set of coefficients for the basis functions in this representation, and by a suitable rewriting of the PDEs, we can obtain algebraic equations for these coefficients.

With both FDMs and FEMs, the resulting algebraic equation system is usually very large, and in the matrix form, such equation systems are usually very sparse. Both FDM and FEM, therefore, heavily rely on sparse matrix representation for the algebraic linear equations, as discussed in Chapter 10. Most general-purpose frameworks for PDEs are based on FEM, or some variant thereof, as this method solves very general problems on complicated problem domains.

Solving PDE problems can be far more resource-demanding compared to other types of computational problems covered so far (e.g., compared to ODEs). It can be resource-demanding partly because of the number of points required to discretize a region of space scale exponentially with the number of dimensions. If a one-dimensional problem requires 100 points to describe, a two-dimensional problem with similar resolution requires $100^2 = 10^4$ points, and a three-dimensional problem requires $100^3 = 10^6$ points. Since each point in the discretized space corresponds to an unknown variable, it is easy to imagine that PDE problems can result in very large equation systems. Defining PDE problems programmatically can also be complicated. One reason is that the possible forms of a PDE vastly outnumber the more limited possible forms of ODEs. Another reason is geometry: while an interval in one-dimensional space is uniquely defined by two points, an area in two-dimensional problems and a volume in three-dimensional problems can have arbitrarily complicated geometries enclosed by curves and surfaces. Defining the problem domain of a PDE and its discretization in a mesh of coordinate points can, therefore, require advanced tools, and there is a large amount of freedom in how boundary conditions can be defined as well. In contrast to ODE problems, there is no standard form to define any PDE problem.

For these reasons, the PDE solvers for Python are only available through libraries and frameworks that are specifically dedicated to PDE problems. For Python, there are at least three significant libraries for solving PDE problems using the FEM method: the FiPy library, the SfePy library, and the FEniCS library. These libraries are extensive and feature-rich, and going into the details of using either is beyond this book's scope. Here, we can only briefly introduce PDE problems and survey prominent examples of PDE

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_11

libraries that can be used from Python. The chapter goes through a few examples that illustrate some of the features of one of these libraries (FEniCS). The hope is that this can give the reader interested in solving PDE problems with Python a bird's-eye overview of the available options and some useful pointers on where to look for further information.

Importing Modules

Basic numerical and plotting usage requires the NumPy and Matplotlib libraries. For 3D plotting, we must explicitly import the `mplot3d` module from the `mpl_toolkits` Matplotlib toolkit library. As usual, assume that these libraries are imported in the following manner.

```
In [1]: import numpy as np
In [2]: import matplotlib.pyplot as plt
In [3]: import matplotlib as mpl
In [4]: import mpl_toolkits.mplot3d
```

Let's also use the `linalg` and the `sparse` modules from SciPy, and to use the `linalg` submodule of the `sparse` module, we need to import it explicitly.

```
In [5]: import scipy.sparse as sp
In [6]: import scipy.sparse.linalg
In [7]: import scipy.linalg as la
```

With these imports, we can access the dense linear algebra module as `la`, while the sparse linear algebra module is accessed as `sp.linalg`. Furthermore, later in this chapter, we use the FEniCS FEM framework, which requires that its `dolfin` and `mshr` libraries be imported in the following manner.

```
In [8]: import dolfin
In [9]: import mshr
```

Partial Differential Equations

The unknown quantity in a PDE is a multivariate function, here denoted as *u*. In an N-dimensional problem, the *u* function depends on *n*-independent variables: $u(x_1, x_2, ..., x_n)$. A general PDE can formally be written as

$$F\left(x_1,x_2,\ldots,x_n,u,\left\{\frac{\partial u}{\partial x_{i_1}}\right\}_{1\le i_1\le n},\left\{\frac{\partial^2 u}{\partial x_{i_1}\partial x_{i_2}}\right\}_{1\le i_1,i_2\le n},\ldots\right)=0, x\in\Omega$$

where $\left\{\frac{\partial u}{\partial x_{i_1}}\right\}_{1\le i_1\le n}$ denotes all first-order derivatives with respect to the independent variables $x_1,\ldots,x_n$, $\left\{\frac{\partial^2 u}{\partial x_{i_1}\partial x_{i_2}}\right\}_{1\le i_1,i_2\le n}$ denotes all second-order derivatives, and so on. Here *F* is a known function that describes the form of the PDE, and Ω is the domain of the PDE problem. Many PDEs that occur in practice only contain up to second-order derivatives, and we typically deal with problems in two or three spatial dimensions and possibly time. When working with PDEs, it is common to simplify the notation by denoting the partial derivatives with respect to an independent variable *x* using the subscript notation: $u_x = \frac{\partial u}{\partial x}$.

Higher-order derivatives are denoted with multiple indices: $u_{xx} = \frac{\partial^2 u}{\partial x^2}$, $u_{xy} = \frac{\partial^2 u}{\partial x \partial y}$, and so on. An example of a typical PDE is the heat equation, which in a two-dimensional Cartesian coordinate system takes the form $u_t = \alpha(u_{xx} + u_{yy})$. Here, the $u = u(t, x, y)$ function describes the temperature at the spatial point (x, y) at time t, and α is the thermal diffusivity coefficient.

To fully specify a particular solution to a PDE, we need to define its boundary conditions, which are known values of the function or a combination of its derivatives along the boundary of the problem domain Ω, as well as the initial values if the problem is time-dependent. The boundary is often denoted as Γ or $\partial\Omega$, and different boundary conditions can generally be given for different parts of the boundary. Two important types of boundary conditions are Dirichlet boundary conditions, which specify the value of the function at the boundary, $u(\boldsymbol{x}) = h(\boldsymbol{x})$ for $\boldsymbol{x} \in \Gamma_D$, and Neumann boundary conditions, which specify the normal derivative on the boundary, $\frac{\partial u(x)}{\partial n} = g(x)$ for $\boldsymbol{x} \in \Gamma_N$, where $\boldsymbol{n}$ is the outward normal from the boundary. Here $h(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are arbitrary functions.

Finite-Difference Methods

The basic idea of the finite-difference method is to approximate the derivatives that occur in a PDE with their finite-difference formulas on a discretized space. For example, the finite-difference formula for the ordinary derivative $\frac{du(x)}{dx}$ on a discretization of the continuous variable x into discrete points $\{x_n\}$ can be approximated with the forward difference formula $\frac{du(x_n)}{dx} \approx \frac{u(x_{n+1}) - u(x_n)}{x_{n+1} - x_n}$, the backward difference formula $\frac{du(x_n)}{dx} \approx \frac{u(x_n) - u(x_{n-1})}{x_n - x_{n-1}}$, or the centered difference formula $\frac{du(x_n)}{dx} \approx \frac{u(x_{n+1}) - u(x_{n-1})}{x_{n+1} - x_{n-1}}$. Similarly, we can construct finite-difference formulas for higher-order derivatives, such as the second-order derivative $\frac{d^2u(x_n)}{dx^2} \approx \frac{u(x_{n+1}) - 2u(x_n) + u(x_{n-1})}{(x_n - x_{n-1})^2}$. Assuming that the discretization of the continuous variable x into discrete points is fine enough, these finite-difference formulas can give good approximations of the derivatives. Replacing derivatives in an ODE or PDE with their finite-difference formulas recasts the equations from differential equations to algebraic equations. If the original ODE or PDE is linear, the algebraic equations are also linear and can be solved with standard linear algebra methods.

To make this method more concrete, consider the ODE problem $u_{xx} = -5$ in the interval $x \in [0, 1]$ and with boundary conditions $u(x = 0) = 1$ and $u(x = 1) = 2$, which, for example, arises from the steady-state heat equation in one dimension. In contrast to the ODE initial value problem considered in Chapter 9, this is a boundary value problem because the value of u is specified at $x = 0$ and $x = 1$. The methods for initial value problems are, therefore, not applicable here. Instead, we can treat this problem by dividing the interval [0, 1] into discrete points x_n, and the problem is then to find the $u(x_n) = u_n$ function at these points. Writing the ODE problem in finite-difference form gives an equation $(u_{n-1} - 2u_n + u_{n+1})/\Delta x^2 = -5$ for every interior point n and the boundary conditions $u_0 = 1$ and $u_{N+1} = 2$. Here, the interval [0, 1] is discretized into $N + 2$ evenly spaced points, including the boundary points, with separation $\Delta x = 1/(N + 1)$. Since the function is known at the two boundary points, there are N unknown variables u_n corresponding to the function values at the interior points. The set of equations for the interior points can be written in a matrix form as $Au = b$, where $u = [u_1, ..., u_N]^T$, $b = \left[-5 - \frac{u_0}{\Delta x^2}, -5, \ldots, -5, -5 - \frac{u_{N+1}}{\Delta x^2}\right]^T$, and

$$A = \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & \dots \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & \ddots \\ \vdots & \vdots & 0 & \ddots & \ddots \end{bmatrix}.$$

Here, the matrix A describes the coupling of the equations for u_n to values at neighboring points due to the finite-difference formula used to approximate the second-order derivative in the ODE. The boundary values are included in the b vector, which also contains the constant right-hand side of the original ODE (the source term). At this point, we can straightforwardly solve the linear equation system $Au = b$ for the unknown vector of u and thereby obtain the approximate values of the $u(x)$ function at the discrete points $\{x_n\}$.

In Python code, we can set up and solve this problem. First, define variables for the number of interior points N, the values of the function at the boundaries u0 and u1, and the spacing between neighboring points dx.

```
In [10]: N = 5
In [11]: u0, u1 = 1, 2
In [12]: dx = 1.0 / (N + 1)
```

Next, construct the matrix A as described in the preceding section. For this, we can use the eye function from NumPy, which creates a two-dimensional array with ones on the diagonal or the upper or lower diagonal that is shifted from the main diagonal by the number given by the k argument.

```
In [13]: A = (np.eye(N, k=-1) - 2 * np.eye(N) + np.eye(N, k=1)) / dx**2
In [14]: A
Out[14]: array([[-72.,  36.,   0.,   0.,   0.],
                [ 36., -72.,  36.,   0.,   0.],
                [  0.,  36., -72.,  36.,   0.],
                [  0.,   0.,  36., -72.,  36.],
                [  0.,   0.,   0.,  36., -72.]])
```

Next, we need to define an array for the vector b, which corresponds to the source term –5 in the differential equation and the boundary condition. The boundary conditions enter the equations via the finite-difference expressions for the derivatives of the first and the last equation (for u_1 and u_N). But these terms are missing from the expression represented by the matrix A and must, therefore, be added to the vector b.

```
In [15]: b = -5 * np.ones(N)
    ...: b[0] -= u0 / dx**2
    ...: b[N-1] -= u1 / dx**2
```

Once the matrix A and the vector b are defined, we can solve the equation system using the linear equation solver from SciPy (we could also use the one provided by NumPy, `np.linalg.solve`).

```
In [16]: u = la.solve(A, b)
```

This completes the solution of this ODE problem. To visualize the solution, here we first create an array x that contains the discrete coordinate points for which we have solved the problem, including the boundary points, and we also create an array U that combines the boundary values and the interior points in one array. The result is then plotted and shown in Figure 11-1.

```
In [17]: x = np.linspace(0, 1, N+2)
In [18]: U = np.hstack([[u0], u, [u1]])
In [19]: fig, ax = plt.subplots(figsize=(8, 4))
    ...: ax.plot(x, U)
    ...: ax.plot(x[1:-1], u, 'ks')
    ...: ax.set_xlim(0, 1)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
    ...: ax.set_ylabel(r"$u(x)$", fontsize=18)
```

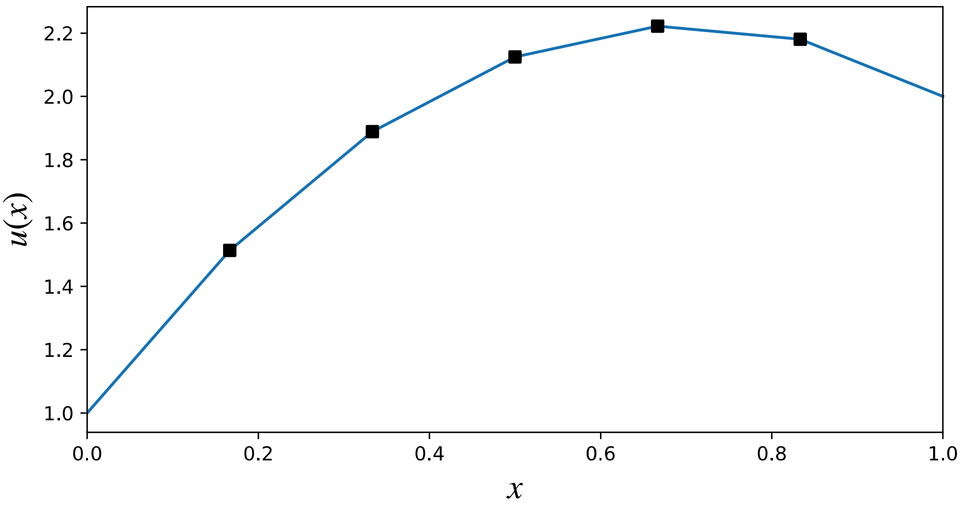

Figure 11-1. *The solution to the second-order ODE boundary value problem introduced in the text*

The finite-difference method can easily be extended to higher dimensions using the finite-difference formula along each discretized coordinate. For a two-dimensional problem, there is a two-dimensional array u for the unknown interior function values, and when using the finite differential formula, obtain a system of coupled equations for the elements in u. We can rearrange the u array into a vector and assemble the corresponding matrix A from the finite-difference equations to write these equations in the standard matrix-vector form.

As an example, consider the following two-dimensional generalization of the previous problem: $u_{xx}+u_{yy} = 0$, with the boundary conditions $u(x = 0) = 3$, $u(x = 1) = -1$, $u(y = 0) = -5$, and $u(y = 1) = 5$. There is no source term here, but the boundary conditions in a two-dimensional problem are more complicated than in the onedimensional problem solved earlier. In finite-difference form, we can write the PDE as $(u_{m-1,n} - 2u_{m,n} + u_{m+1,n})/x^2 + (u_{m,n-1} - 2u_{m,n} + u_{m,n+1})/y^2 = 0$. If we divide the x and y intervals into N interior points ($N + 2$ points including the boundary points), then $\Delta x = \Delta y = \frac{1}{N+1}$, and u is an $N \times N$ matrix. To write the equation in the standard form $Av = b$, we can rearrange the matrix u by stacking its rows or columns into a vector of size $N^2 \times 1$. The matrix A is then of size $N^2 \times N^2$, which can be very big if we need to use a fine discretization of the x and y coordinates. For example, using 100 points along both x and y gives an equation system that has 10^4 unknown values u_{mn}, and the matrix A has $100^4 = 10^8$ elements. Fortunately, since the finite-difference formula only couples neighboring points, the matrix A turns out to be very sparse, and here, we can benefit significantly from working with sparse matrices, as shown in the following.

To solve this PDE problem with Python and the finite-element method, start by defining variables for the number of interior points and the values along the four boundaries of the unit square.

```
In [20]: N = 100
In [21]: u0_t, u0_b = 5, -5
In [22]: u0_l, u0_r = 3, -1
In [23]: dx = 1. / (N+1)
```

This also computed the separation dx between the uniformly spaced coordinate points in the discretization of *x* and *y* (assumed equal). Because the finite-difference formula couples both neighboring rows and columns, it is slightly more involved to construct the matrix A for this example. However, a relatively direct approach is first to define the matrix A_1d corresponding to the one-dimensional formula along one of the coordinates (say *x* or the index *m* in $u_{m,n}$). To distribute this formula along each row, we can take the tensor product of the identity matrix of size $N \times N$ with the A_1d matrix. The result describes all derivatives along the m-index for all indices n. To cover the terms that couple the equation for $u_{m,n}$ to $u_{m,n+1}$ and $u_{m,n-1}$, that is, the derivatives along the index n, we need to add diagonals separated from the main diagonal by *N* positions. The following steps construct *A* using the eye and kron functions from the scipy.sparse module. The result is a sparse matrix A that describes the finite-difference equation system for the two-dimensional PDE considered here.

```
In [24]: A_1d = (sp.eye(N, k=-1) + sp.eye(N, k=1) - 4 * sp.eye(N))/dx**2
In [25]: A = sp.kron(sp.eye(N), A_1d) + \
    ...:     (sp.eye(N**2, k=-N) + sp.eye(N**2, k=N))/dx**2
In [26]: A
Out[26]: <10000x10000 sparse matrix of type '<type 'numpy.float64'>'
          with 49600 stored elements in Compressed Sparse Row format>
```

The printout of A shows that it is a sparse matrix with 10^8 elements with 49,600 nonzero elements, so only 1 out of about 2000 elements is nonzero, and *A* is very sparse. To construct the vector *b* from the boundary conditions, it is convenient to create an $N \times N$ array of zeros and assign the boundary condition to edge elements of this array (which are the corresponding elements in u that are coupled to the boundaries—i.e., the interior points that are neighbors to the boundary). Once this $N \times N$ array is created and assigned, we can use the reshape method to rearrange it into an $N^2 \times 1$ vector that can be used in the $Av = b$ equation.

```
In [27]: b = np.zeros((N, N))
    ...: b[0, :] += u0_b   # bottom
    ...: b[-1, :] += u0_t  # top
    ...: b[:, 0] += u0_l   # left
    ...: b[:, -1] += u0_r  # right
    ...: b = - b.reshape(N**2) / dx**2
```

When the A and b arrays are created, we can proceed to solve the equation system for the vector v and use the reshape method to arrange it back into the $N \times N$ matrix u.

```
In [28]: v = sp.linalg.spsolve(A, b)
In [29]: u = v.reshape(N, N)
```

For plotting purposes, create a matrix U that combines the u matrix with the boundary conditions. With the coordinate matrices X and Y, plot a colormap graph and a 3D surface view of the solution. The result is shown in Figure 11-2.

```
In [30]: U = np.vstack([np.ones((1, N+2)) * u0_b,
    ...:                np.hstack([np.ones((N, 1)) * u0_l, u,
    ...:                           np.ones((N, 1)) * u0_r]),
    ...:                np.ones((1, N+2)) * u0_t])
In [31]: x = np.linspace(0, 1, N+2)
In [32]: X, Y = np.meshgrid(x, x)
In [33]: fig = plt.figure(figsize=(12, 5.5))
    ...: cmap = mpl.cm.get_cmap('RdBu_r')
    ...:
    ...: ax = fig.add_subplot(1, 2, 1)
    ...: c = ax.pcolor(X, Y, U, vmin=-5, vmax=5, cmap=cmap)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...:
    ...: ax = fig.add_subplot(1, 2, 2, projection='3d')
    ...: p = ax.plot_surface(X, Y, U, vmin=-5, vmax=5, rstride=3, cstride=3,
    ...:                     linewidth=0, cmap=cmap)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: cb = plt.colorbar(p, ax=ax, shrink=0.75)
    ...: cb.set_label(r"$u(x_1, x_2)$", fontsize=18)
```

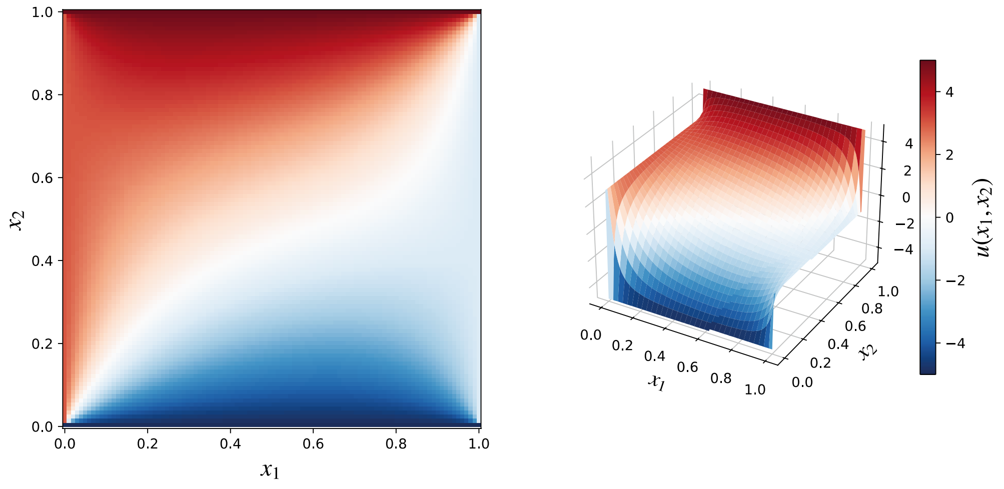

Figure 11-2. *The solution to the two-dimensional heat equation with Dirichlet boundary conditions defined in the text*

As mentioned, FDM methods result in matrices A that are very sparse, and using sparse matrix data structures, such as those provided by `scipy.sparse`, can give significant performance improvements compared to using dense NumPy arrays. To illustrate the importance of using sparse matrices for this problem, we can compare the time required for solving the $Av = b$ equation using the IPython command `%timeit`, for the two cases where A is a sparse and a dense matrix.

```
In [34]: A_dense = A.todense()
In [35]: %timeit la.solve(A_dense, b)
1 loops, best of 3: 10.8 s per loop
In [36]: %timeit sp.linalg.spsolve(A, b)
10 loops, best of 3: 31.9 ms per loop
```

These results show that using sparse matrices for the present problem results in a speedup of several orders of magnitude (this case has a speedup of a factor $10.8/0.0319 \approx 340$).

The finite-difference method used in the last two examples is a powerful and relatively simple method for solving ODE boundary value problems and PDE problems with simple geometries. However, it is not easily adapted to problems on more complicated domains or problems on nonuniform coordinate grids. For such problems, finite-element methods are typically more flexible and convenient. Although FEMs are conceptually more complicated than FDMs, they can be computationally efficient and adapt well to complex problem domains and more involved boundary conditions.

Finite-Element Methods

The finite-element method is a powerful and universal way to convert PDEs into algebraic equations. The basic idea of this method is to represent the domain on which the PDE is defined with a finite set of discrete regions, or *elements*, and to approximate the unknown function as a linear combination of basis functions with local support on each of these elements (or on a small group of neighboring elements). Mathematically, this approximation solution, u_h, represents a projection of the exact solution u in the function space V (e.g., continuous real-valued functions) onto a finite subspace $V_h \subset V$ related to the discretization of the problem domain. If V_h is a suitable subspace of V, then it can be expected that u_h can be a good approximation to u.

To solve the approximate problem on the simplified function space V_h, we can first rewrite the PDE from its original formulation, known as the *strong form*, to its corresponding variational form, also known as the *weak form*. To obtain the weak form, multiply the PDE with an arbitrary function v and integrate it over the entire problem domain. The v function is called a *test function*, and it can generally be defined on function space that differs from V and V_h.

For example, consider the steady-state heat equation (also known as the Poisson equation) solved using the FDM earlier in this chapter: the strong form of this equation is $-\Delta u(\boldsymbol{x}) = f(\boldsymbol{x})$, where we have used the vector operator notation. Multiplying this equation with a test function v and integrating over the domain $\boldsymbol{x} \in \Omega$ obtains the weak form:

$$-\int_\Omega \Delta u \;v \;\mathrm{d}x = \int_\Omega fv \;\mathrm{d}x.$$

Since the exact solution u satisfies the strong form, it also satisfies the weak form of the PDE for any reasonable choice of v. The reverse does not necessarily hold true. But if a function, u_h, called a *trial function* in this context, satisfies the weak form for a large class of suitably chosen test functions v, then it is plausible that it is a good approximation to the exact solution u.

To treat this problem numerically, we first need to make the transition from the infinite-dimensional function spaces V and $\hat{V}$ to approximate finite-dimensional function spaces V_h and $\hat{V}_h$:

$$-\int_\Omega \Delta u_h v_h \;\mathrm{d}x = \int_\Omega f v_h \;\mathrm{d}x,$$

where $u_h \in V_h$ and $v_h \in \hat{V}_h$. The key point here is that V_h and $\hat{V}_h$ are finite-dimensional, so we can use a finite set of basis functions $\{\phi_i\}$ and $\left\{\hat{\phi}_i\right\}$ that spans the function spaces V_h and $\hat{V}_h$, respectively, to describe the functions u_h and v_h. We can express u_h as a linear combination of the basis functions that span its function space, $u_h = \sum U_i\phi_i$. Inserting this linear combination in the weak form of the PDE and carrying out the integrals and differential operators on the basis functions instead of directly over terms in the PDE yields a set of algebraic equations.

To obtain an equation system on the simple form $AU = b$, we also must write the weak form of the PDE in bilinear form with respect to the u_h and v_h functions $a(u_h, v_h) = L(v_h)$, for some functions a and L. This is not always possible, but for the current example of the Poisson equation, we can obtain this form by integrating by parts:

$$-\int_\Omega \Delta u_h v_h \ \mathrm{d}x = \int_\Omega \nabla u_h . \nabla v_h \ \mathrm{d}x - \int_\Omega \nabla.(\nabla u_h v_h)\mathrm{d}x = \int_\Omega \nabla u_h . \nabla v_h \ \mathrm{d}x - \int_{\partial\Omega} (\nabla u_h . n) v_h \ d\Gamma.$$

where in the second equality, we have also applied Gauss' theorem to convert the second term to an integral over the boundary $\partial\Omega$ of the domain Ω. Here $\boldsymbol{n}$ is the outward normal vector of the boundary $\partial\Omega$. There is no general method for rewriting a PDE from the strong form to the weak form, and each problem must be approached on a case-by-case basis. However, the technique used here, to integrate by part and rewrite the resulting integrals using integral identities, can be used for many frequently occurring PDEs.

We must also deal with the boundary term in the preceding weak-form equation to reach the bilinear form that can be approached with standard linear algebra methods. To this end, assume that the problem satisfies the Dirichlet boundary condition on the part of $\partial\Omega$ denoted Γ_D and Neumann boundary conditions on the remaining part of $\partial\Omega$, denoted Γ_N: $\{u = h, x \in \Gamma_D\}$ and $\{\nabla u \cdot \boldsymbol{n} = g, x \in \Gamma_\mathrm{N}\}$. Not all boundary conditions are of Dirichlet or Neumann type, but these cover many physically motivated situations.

Since we can choose the test functions v_h, we can let v_h vanish on the part of the boundary that satisfies Dirichlet boundary conditions. In this case, obtain the following weak form of the PDE problem.

$$\int_\Omega \nabla u_h . \nabla v_h \ \mathrm{d}x = \int_\Omega f v_h \ \mathrm{d}x + \int_{\Gamma_\mathrm{N}} g \ v_h \ d\Gamma$$

If we substitute the u_k function for its expression as a linear combination of basis functions and substitute the test function with one of its basis functions, we obtain an algebraic equation.

$$\sum U_j \int_\Omega \nabla \phi_j . \nabla \hat{\phi}_i \ \mathrm{d}x = \int_\Omega f \hat{\phi}_i \ \mathrm{d}x + \int_{\Gamma_\mathrm{N}} g \hat{\phi}_i \ d\Gamma$$

If there are N basis functions in V_k, then there are N unknown coefficients U_i, and we need N-independent test functions $\hat{\phi}_i$ to obtain a closed equation system. This equation system is on the form $AU = b$ with $A_{ij} = \int_\Omega \nabla \phi_j . \nabla \hat{\phi}_i \ \mathrm{d}x$ and $b_i = \int_\Omega f \hat{\phi}_i \ \mathrm{d}x + \int_{\Gamma_\mathrm{N}} g \hat{\phi}_i \ d\Gamma$. Following this procedure we have converted the PDE problem into a system of linear equations that can be readily solved using techniques discussed in previous chapters.

In practice, a very large number of basis functions can be required to obtain a good approximation to the exact solution, and the linear equation system generated by FEMs is, therefore, often very large. However, the fact that each basis function has support only at one or a few nearby elements in the discretization of the problem domain ensures that the matrix A is sparse, which makes it tractable to solve rather large-scale FEM problems. Note that an important property of the basis functions ϕ_i and $\hat{\phi}_i$ is that it should be easy to compute the derivatives and integrals of the expression that occurs in the final weak form of the problem so that the matrix A and vector b can be assembled efficiently. Typical examples of basis functions are low-order polynomial functions that are nonzero only within a single element. Figure 11-3 is a one-dimensional illustration of this type of basis function, where the interval [0, 6] is discretized using five interior points. A continuous function (black solid curve) is approximated as a piecewise linear function (dashed red line) by suitably weighted triangular basic functions (blue solid lines).

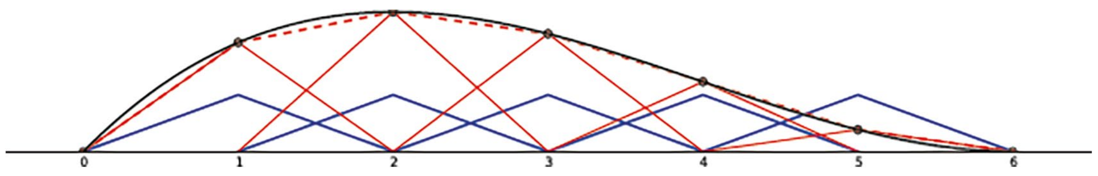

Figure 11-3. *An example of possible basis functions (blue), with local support, for the one-dimensional domain [0, 6]*

When using FEM software for solving PDE problems, it is typically required to convert the PDE to weak form by hand and, if possible, rewrite it on the bilinear form $a(u, v) = L(v)$. It is also necessary to provide a suitable discretization of the problem domain. This discretization is called a mesh, and it is usually made up of triangular partitioning (or their higher-order generalizations) of the total domain. Meshing an intricate problem domain can be a complicated process, and it may require using sophisticated software, especially dedicated to mesh generation. There are tools for programmatically generating meshes for simple geometries, which we see examples of in the following section.

Once a mesh is generated and the PDE problem is written in a suitable weak form, we can feed the problem into a FEM framework, which then automatically assembles the algebraic equation system and applies suitable sparse equation solvers to find the solution. In this process, we often have a choice of what type of basis functions to use and which type of solver to use. Once the algebraic equation is solved, we can construct the approximation solution to the PDE with the help of the basis functions, and we can, for example, visualize the solution or post-process it in some other fashion.

Solving a PDE using FEM typically involves the following steps.

1. Generate a mesh for the problem domain.
2. Write the PDE in weak form.
3. Program the problem in the FEM framework.
4. Solve the resulting algebraic equations.
5. Post-process and/or visualize the solution.

The following section reviews available FEM frameworks that can be used with Python and then look at several examples that illustrate some of the key steps in the PDE solution process using FEM.

Survey of FEM Libraries

For Python, there are at least three significant FEM packages: FiPy, SfePy, and FEniCS. These are all rather full-featured frameworks capable of solving a wide range of PDE problems. Technically, the FiPy library is not FEM software but rather a finite-volume method (FVM) software, but the gist of this method is quite similar to FEM. The FiPy framework can be obtained from www.ctcms.nist.gov/fipy. The SfePy library is a FEM software that takes a slightly different approach to defining PDE problems in that it uses Python files as configuration files for its FEM solver, rather than programmatically setting up a FEM problem (although this mode of operation is technically also supported in SfePy). The SfePy library is available from http://sfepy.org. The third major framework for FEM with Python is FEniCS, which is written for C++ and Python. The FEniCS framework is my personal favorite when it comes to FEM software for Python, as it provides an elegant Python interface to a powerful FEM engine. Like FDM problems, FEM problems typically result in very large-scale equation systems that require using sparse matrix techniques to solve efficiently. A crucial part of a FEM framework is, therefore, to efficiently solve large-scale linear and nonlinear systems, using sparse matrix representation and direct or iterative solvers that work on sparse systems, possibly using parallelization. Each framework mentioned in the preceding section supports multiple backends for such low-level computations. For example, many FEM frameworks can use the PETSc and Trilinos frameworks.

Although I'm not able to comprehensively explain how to use these FEM frameworks here, the following section looks at solving example problems with FEniCS and thereby introduce some of its basic features and usage. The hope is that the examples can give a flavor of how it is to work with FEM problems in Python and provide a starting point for readers interested in learning more about FEM with Python.

Solving PDEs Using FEniCS

This section solves a series of increasingly complicated PDEs using the FEniCS framework and introduces the workflow and a few of the main features of this FEM software. For a thorough introduction to the FEniCS framework, see the documentation on the project websites and the official FEniCS book (Anders Logg, 2012).

■ **FEniCS** FEniCS is a highly capable FEM framework that is made up of a collection of libraries and tools for solving PDE problems. Much of FEniCS is programmed in C++ but provides an official Python interface. Because of the complexity of the many dependencies of the FEniCS libraries to external low-level numerical libraries, FEniCS is usually packaged and installed as an independent environment. However, it can also be installed using conda on some platforms. For more information about the FEniCS, see `http://fenicsproject.org`. At the time of writing, the most recent version is 2019.1.0. A new version called FEniCSx is also under development, but let's use the stable original FEniCS here.

The Python interface to FEniCS is provided by a library named `dolfin`. For mesh generation, let's use the `mshr` library. The following code assumes that these libraries are entirely imported, as shown at the beginning of this chapter. For a summary of these libraries' most important functions and classes, see Table 11-1 and Table 11-2.

Table 11-1. *Summary of Selected Functions and Classes in the dolfin Library*

Function/Class	Description	Example
`parameters`	Dictionary holding configuration parameters for the FEniCS framework	`dolfin.parameters["reorder_dofs_serial"]`
`RectangleMesh`	Object for generating a rectangular 2D mesh	`mesh = dolfin.RectangularMesh(dolfin.Point(0, 0),dolfin.Point(1, 1), 10, 10)`
`MeshFunction`	Function defined over a given mesh	`dolfin.MeshFunction("size_t", mesh, mesh.topology().dim()-1)`
`FunctionSpace`	Object for representing a function space	`V = dolfin.FunctionSpace(mesh, 'Lagrange', 1)`
`TrialFunction`	Object for representing a trial function defined in a given function space	`u  = dolfin.TrialFunction(V)`

(*continued*)

Table 11-1. (*continued*)

Function/Class	Description	Example
`TestFunction`	Object for representing a test function defined in a given function space	`v = dolfin.TestFunction(V)`
`Function`	Object for representing unknown functions appearing in the weak form of a PDE	`u_sol = dolfin.Function(V)`
`Constant`	Object for representing a fixed constant	`c = dolfin.Constant(1.0)`
`Expression`	Representation of a mathematical expression in terms of the spatial coordinates	`dolfin.Expression("x[0]*x[0] + x[1]*x[1]")`
`DirichletBC`	Object for representing Dirichlet-type boundary conditions	`dolfin.DirichletBC(V, u0, u0_boundary)`
`Equation`	Object for representing an equation, for example, generated by using the == operator with other FEniCS objects	`a == L`
`inner`	Symbolic representation of the inner product	`dolfin.inner(u, v)`
`nabla_grad`	Symbolic representation of the gradient operator	`dolfin.nabla_grad(u)`
`dx`	Symbolic representation of the volume measure for integration	`f*v*dx`
`ds`	Symbolic representation of a line measure for integration	`g_v * v * dolfin.ds(0, domain=mesh, subdomain_data=boundary_parts)`
`assemble`	Assemble the algebraic equations by carrying out the integrations over the basis functions.	`A = dolfin.assemble(a)`
`solve`	Solve an algebraic equation.	`dolfin.solve(A, u_sol.vector(), b)`
`plot`	Plot a function or expression.	`dolfin.plot(u_sol)`
`File`	Write a function to a file that can be opened with visualization software such as ParaView.	`dolfin.File('u_sol.pvd') << u_sol`
`refine`	Refine a mesh by splitting a selection of the existing mesh elements into smaller pieces.	`mesh = dolfin.refine(mesh, cell_markers)`
`AutoSubDomain`	Representation of a subset of a domain, selected from all elements by the indicator function passed to it as argument.	`dolfin.AutoSubDomain(v_boundary_func)`

Table 11-2. *Summary of Selected Functions and Classes in the mshr and dolfin Library*

Function/Class	**Description**
`dolfin.Point`	Representation of a coordinate point
`mshr.Circle`	Representation of a geometrical object with the shape of a circle, which can be used to compose 2D domain
`mshr.Ellipse`	Representation of a geometrical object with the shape of an ellipse
`mshr.Rectangle`	Representation of a domain defined by a rectangle in 2D
`mshr.Box`	Representation of a domain defined by a box in 3D
`mshr.Sphere`	Representation of a domain defined by a sphere in 3D
`mshr.generate_mesh`	Generates a mesh from a domain composed of geometrical objects, such as those listed in the preceding section

Before using FEniCS and the `dolfin` Python library, we must set two configuration parameters via the `dolfin.parameters` dictionary to obtain the behavior that we need in the following examples.

```
In [37]: dolfin.parameters["reorder_dofs_serial"] = False
In [38]: dolfin.parameters["allow_extrapolation"] = True
```

Let's begin with FEniCS by reconsidering the steady-state heat equation in two dimensions solved earlier in this chapter using the FDM. Here, consider the problem $u_{xx} + u_{yy} = f$, where f is a source function. To begin, let's assume that the boundary conditions are $u(x = 0, y) = u(x = 1, y) = 0$ and $u(x, y = 0) = u(x, y = 1) = 0$. Later examples demonstrate how to define Dirichlet and Neumann boundary conditions.

The first step in the solution of a PDE with FEM is to define a mesh that describes the discretization of the problem domain. In the current example, the problem domain is the unit square $x,y \in [0, 1]$. For simple geometries like this, there are functions in the `dolfin` library for generating the mesh. Here, we use the `RectangleMesh` function, which, as the first two arguments, takes the coordinate points (x_0, y_0) and (x_1, y_1), represented as `dolfin.Point` instances, where (x_0, y_0) is the coordinates of the lower-left corner of the rectangle and (x_1, y_1) of the upper-right corner. The fifth and sixth arguments are the numbers of elements along the x and y directions, respectively. The resulting mesh object is viewed in a Jupyter Notebook via its rich display system (here, we generate a less fine mesh to display the mesh structure), as shown in Figure 11-4.

```
In [39]: N1 = N2 = 75
In [40]: mesh = dolfin.RectangleMesh(
    ...:    dolfin.Point(0, 0), dolfin.Point (1, 1), N1, N2)
In [41]: dolfin.RectangleMesh(
    ...:     dolfin.Point(0, 0), dolfin.Point(1, 1), 10, 10) # for display
```

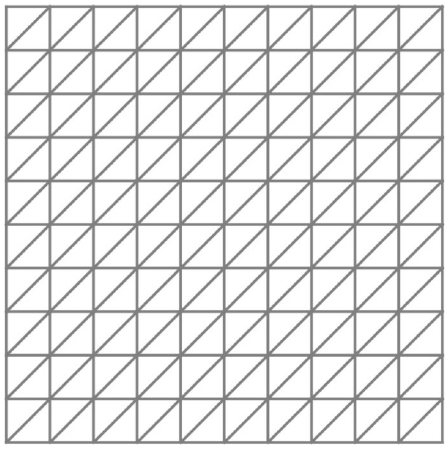

Figure 11-4. *A rectangular mesh generated using dolfin.RectangleMesh*

This mesh for the problem domain is the key to discretizing the problem into a form that can be treated using numerical methods. The next step is to define a representation of the function space for the trial and the test functions, utilizing the `dolfin.FunctionSpace` class. The constructor of this class takes at least three arguments: a mesh object, the name of the type of basis function, and the degree of the basis function. Let's use the Lagrange type of basis functions of degree one (linear basis functions).

```
In [42]: V = dolfin.FunctionSpace(mesh, 'Lagrange', 1)
```

Once the mesh and the function space objects are created, we need to create objects for the trial function u_h and the test function v_h, which we can use to define the weak form of the PDE of interest. In FEniCS, let's use the `dolfin.TrialFunction` and `dolfin.TestFunction` classes for this purpose. They both require a function space object as the first argument to their constructors.

```
In [43]: u = dolfin.TrialFunction(V)
In [44]: v = dolfin.TestFunction(V)
```

The purpose of defining representations of the function space `V` and the trial and test functions `u` and `v` is to be able to construct a representation of a generic PDE on the weak form. For the steady-state heat equation that we are studying here, the weak form was shown in the previous section to be (in the absence of Neumann boundary conditions)

$$\int_\Omega \nabla u . \nabla v \ \mathrm{d}x = \int_\Omega fv \ \mathrm{d}x.$$

Arriving at this form usually requires rewriting and transforming the direct integrals over the PDE by hand, typically by performing integration by parts. In FEniCS, the PDE itself is defined using the integrands that appear in the weak form, including the integral measure (i.e., the dx). To this end, the `dolfin` library provides several functions acting on the trial and test function objects `v` and `u` that represent operations on these functions that commonly occur in the weak form of a PDE. For example, in the present case, the integrand of the left-hand side integral is $\nabla u \cdot \nabla v \, \mathrm{d}x$. To represent this expression, we need a symbolic representation of the inner product, the gradients of u and v, and the integration measure dx. The names for these functions in the `dolfin` library are `inner`, `nabla_grad`, and `dx`, respectively, and using these functions, we can create a representation of $a(u, v) = \nabla u \cdot \nabla v \, \mathrm{d}x$ that the FEniCS framework understands and can work with.

```
In [45]: a = dolfin.inner(dolfin.nabla_grad(u), dolfin.nabla_grad(v)) * dolfin.dx
```

Likewise, we need a representation of $b(v) = fv \, dx$ for the right-hand side. At this point, we need to specify an explicit form of f (the source term in the original PDF) to proceed with the solution of the problem. Here, let's look at two types of functions: $f = 1$ (a constant) and $f = x^2 + y^2$ (a function of x and y). To represent $f = 1$, we can use the `dolfin.Constant` object. It takes the value of the constant that it represents as its only argument.

```
In [46]: f1 = dolfin.Constant(1.0)
In [47]: L1 = f1 * v * dolfin.dx
```

If f is a function of x and y, we need to use the `dolfin.Expression` object to represent f. The constructor of this object takes a string as the first argument containing an expression corresponding to the function. This expression must be defined in C++ syntax, since the FEniCS framework automatically generates and compiles a C++ function to efficiently evaluate the expression. In the expression, we have access to a variable `x`, which is an array of coordinates at a specific point, where x is accessed as `x[0]`, y as `x[1]`, and so on. For example, to write the expression for $f(x, y) = x^2 + y^2$, we can use `"x[0]*x[0] + x[1]*x[1]"`. Note that because we need to use C++ syntax in this expression, we can*not* use the Python syntax `x[0]**2`. The `Expression` class also takes the keyword argument `degree` that specifies the degree of the basis function or the keyword argument `element` that describes the finite elements, which, for example, can be obtained using the `ufl_element` method of the function space object `V`.

```
In [48]: f2 = dolfin.Expression("x[0]*x[0] + x[1]*x[1]", degree=1)
In [49]: L2 = f2 * v * dolfin.dx
```

At this point, we have defined symbolic representations of the terms that occur in the weak form of the PDE. The next step is to define the boundary conditions. Let's begin with a simple uniform Dirichlet-type boundary condition. The `dolfin` library contains a class `DirichletBC` for representing this type of boundary conditions. We can use this class to represent arbitrary functions along the boundaries of the problem domain, but in this first example, consider the simple boundary condition $u = 0$ on the entire boundary. To represent the constant value on the boundary (zero in this case), we can again use the `dolfin.Constant` class.

```
In [50]: u0 = dolfin.Constant(0)
```

In addition to the boundary condition value, we also need to define a function (here called `u0_boundary`) used to select different parts of the boundary when creating an instance of the `DirichletBC` class. This function takes two arguments: a coordinate array `x` and a flag `on_boundary` that indicates if a point is on the physical boundary of the mesh, and it should return `True` if the point `x` belongs to the boundary and `False` otherwise. Since this function is evaluated for every vertex in the mesh, by customizing the function, we could pin down the function value at arbitrary parts of the problem domain to specific values or expressions. However, here, we only need to select all the points on the physical boundary, so we can simply let the `u0_boundary` function return the `on_boundary` argument.

```
In [51]: def u0_boundary(x, on_boundary):
    ...:     return on_boundary
```

Once we have an expression for the value on the boundary, u0, and a function for selecting the boundary from the mesh vertices, u0_boundary, we can, with the function space object V, finally create the DirichletBC object.

```
In [52]: bc = dolfin.DirichletBC(V, u0, u0_boundary)
```

This completes the specification of the PDE problem, and our next step is to convert the problem into an algebraic form by assembling the matrix and vector from the weak form representations of the PDE. We can do this explicitly using the dolfin.assemble function.

```
In [53]: A = dolfin.assemble(a)
In [54]: b = dolfin.assemble(L1)
In [55]: bc.apply(A, b)
```

This results in a matrix *A* and vector *b* that define the algebraic equation system for the unknown function. The apply method of the DirichletBC class instance bc, which modifies the A and b objects so that the boundary condition is accounted for in the equations, was used here.

To finally solve the problem, we need to create a function object for storing the unknown solution and call the dolfin.solve function, providing the A matrix and the b vector, as well as the underlying data array of a Function object. We can obtain the data array for a Function instance by calling the vector method on the object.

```
In [56]: u_sol1 = dolfin.Function(V)
In [57]: dolfin.solve(A, u_sol1.vector(), b)
```

Here, the Function object for the solution u_sol1, and the call to dolfin.solve function solves the equation system and fills in the values in the data array of the u_sol1 object. Here, the PDE problem was solved by explicitly assembling the A and b matrices and passing the results to the dolfin.solve function. These steps can also be carried out automatically by the dolfin.solve function, by passing a dolfin.Equation object as the first argument to the function, the Function object for the solution as the second argument, and a boundary condition (or list of boundary conditions) as the third argument. We can create the Equation object using, for example, a == L2.

```
In [58]: u_sol2 = dolfin.Function(V)
In [59]: dolfin.solve(a == L2, u_sol2, bc)
```

This is slightly more concise than the method used to find u_sol1 using the equivalence of a == L1. But in some cases, when a problem needs to be solved for multiple situations, it can be useful to use explicit assembling of the matrix A and/or the vector b, so it is worthwhile to be familiar with both methods.

With the solution available as a FEniCS Function object, we can proceed with post-processing and visualizing the solution in many ways. A straightforward way to plot the solution is to use the built-in dolfin.plot function, which can be used to plot mesh objects, function objects, and other objects (see the docstring for dolfin.plot for more information). For example, to plot the solution u_sol2, call dolfin.plot(u_sol2). The resulting graph window is shown in Figure 11-5.

```
In [60]: dolfin.plot(u_sol2)
```

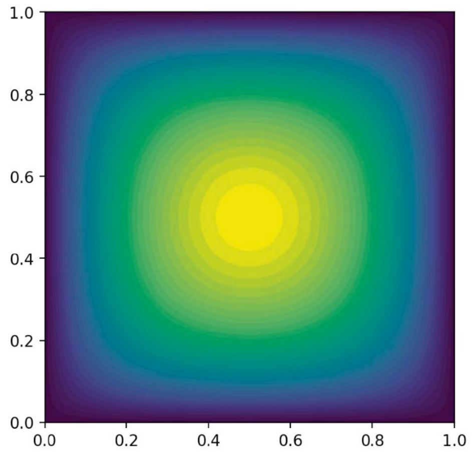

Figure 11-5. *A graph of the mesh function u_sol2, produced by the plot function in the dolfin library*

Using `dolfin.plot` is a good way of quickly visualizing a solution or a grid. But for better control of the visualization, it is often necessary to export the data and plot it in dedicated visualization software, such as ParaView.[1] To save the solutions `u_sol1` and `u_sol2` in a format that can be opened with ParaView, we can use the `dolfin.File` object to generate PVD files (collections of VTK files) and append objects to the file using the `<<` operator in a C++ stream-like fashion.

```
In [61]: dolfin.File('u_sol1.pvd') << u_sol1
```

We can also add multiple objects to a PVD file using this method.

```
In [62]: f = dolfin.File('u_sol_and_mesh.pvd')
    ...: f << mesh
    ...: f << u_sol1
    ...: f << u_sol2
```

Exporting data for FEniCS objects to files that can be loaded and visualized with external visualization software is a method that benefits from the many advantages of powerful visualization software, such as interactivity, parallel processing, and high level of control of the visualizations, to mention a few. However, in many cases, it might be preferable to work within, for example, the Jupyter Notebook for visualization of the solutions and the mesh. For relatively simple problems in one, two, and, to some extent, three dimensions, we can use Matplotlib to visualize meshes and solution functions directly. To be able to use Matplotlib, we need to obtain a NumPy array with data corresponding to the FEniCS function object. There are several ways to construct such arrays. The FEniCS function object can be called like a function with an array (list) of coordinate values.

```
In [63]: u_sol1([0.21, 0.67])
Out[63]: 0.0466076997781351
```

[1] http://www.paraview.org

This allows us to evaluate the solution at arbitrary points within the problem domain. We can also obtain the values of a function object like `u_sol1` at the mesh vertices as a FEniCS vector using the `vector` method, which can be converted to a NumPy array using the `np.array` function. The resulting NumPy arrays are flat (one-dimensional). For a two-dimensional rectangular mesh (like in the current example), it is sufficient to reshape the flat array to obtain a two-dimensional array that can be plotted with, for example, the `pcolor`, `contour`, or `plot_surface` functions from Matplotlib. The following steps convert the underlying data of the `u_sol1` and `u_sol2` function objects to NumPy arrays, which are then plotted using Matplotlib. The result is shown in Figure 11-6.

```
In [64]: u_mat1 = np.array(u_sol1.vector()).reshape(N1+1, N2+1)
In [65]: u_mat2 = np.array(u_sol2.vector()).reshape(N1+1, N2+1)
In [66]: X, Y = np.meshgrid(np.linspace(0, 1, N1+2), np.linspace(0, 1, N2+2))
In [67]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
    ...:
    ...: c = ax1.pcolor(X, Y, u_mat1, cmap=mpl.cm.get_cmap('Reds'))
    ...: cb = plt.colorbar(c, ax=ax1)
    ...: ax1.set_xlabel(r"$x$", fontsize=18)
    ...: ax1.set_ylabel(r"$y$", fontsize=18)
    ...: cb.set_label(r"$u(x, y)$", fontsize=18)
    ...: cb.set_ticks([0.0, 0.02, 0.04, 0.06])
    ...:
    ...: c = ax2.pcolor(X, Y, u_mat2, cmap=mpl.cm.get_cmap('Reds'))
    ...: cb = plt.colorbar(c, ax=ax2)
    ...: ax1.set_xlabel(r"$x$", fontsize=18)
    ...: ax1.set_ylabel(r"$y$", fontsize=18)
    ...: cb.set_label(r"$u(x, x)$", fontsize=18)
    ...: cb.set_ticks([0.0, 0.02, 0.04])
```

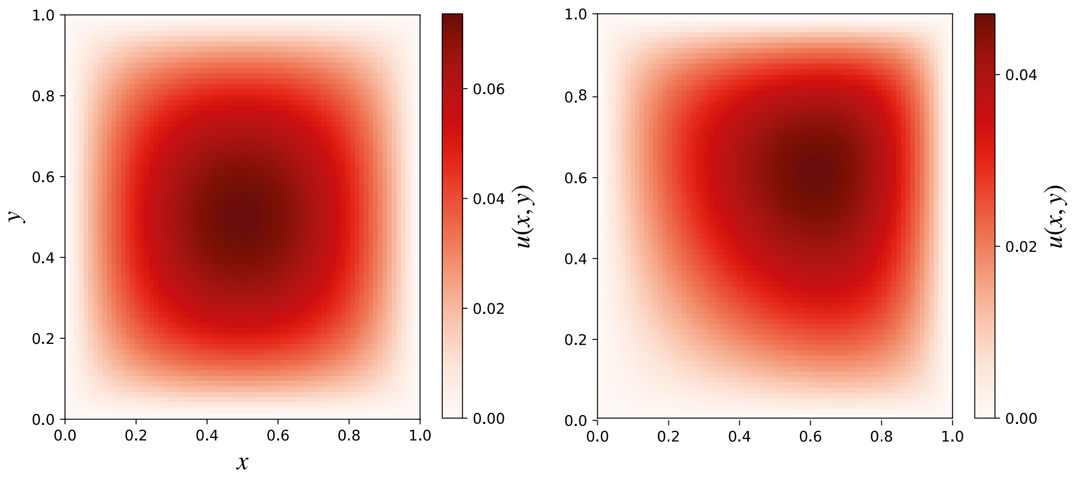

***Figure 11-6.** The solution of the steady-state heat equation on the unit square, with source terms $f = 1$ (left) and $f = x^2 + y^2$ (right), subject to the condition that the $u(x, y)$ function is zero on the boundary*

The method used to produce Figure 11-6 is simple and convenient but only works for rectangular meshes. For more complicated meshes, the vertex coordinates are not organized structurally, and a simple reshaping of the flat array data is not sufficient. However, the `Mesh` object representing the mesh for the problem domain contains a list of the coordinates for each vertex. Together with values from a `Function` object, these can be combined into a form plotted with Matplotlib `triplot` and `tripcolor` functions. To use these plot functions, we first need to create a `Triangulation` object from the vertex coordinates for the mesh.

```
In [68]: coordinates = mesh.coordinates()
    ...: triangles = mesh.cells()
    ...: triangulation = mpl.tri.Triangulation(
    ...:     coordinates[:, 0], coordinates[:, 1], triangles)
```

With the triangulation object defined, we can directly plot the array data for FEniCS functions using `triplot` and `tripcolor`, as shown in the following code. The resulting graph is shown in Figure 11-7.

```
In [69]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
    ...: ax1.triplot(triangulation)
    ...: ax1.set_xlabel(r"$x$", fontsize=18)
    ...: ax1.set_ylabel(r"$y$", fontsize=18)
    ...: cmap = mpl.cm.get_cmap('Reds')
    ...: c = ax2.tripcolor(triangulation, np.array(u_sol2.vector()), cmap=cmap)
    ...: cb = plt.colorbar(c, ax=ax2)
    ...: ax2.set_xlabel(r"$x$", fontsize=18)
    ...: ax2.set_ylabel(r"$y$", fontsize=18)
    ...: cb.set_label(r"$u(x, y)$", fontsize=18)
    ...: cb.set_ticks([0.0, 0.02, 0.04])
```

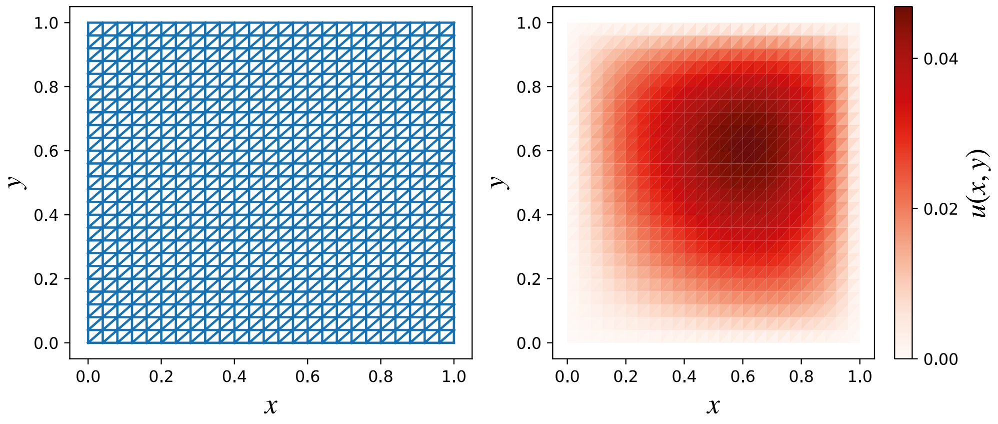

Figure 11-7. *The same as Figure 11-6, except that this graph was produced with Matplotlib's triangulation functions. The mesh is plotted to the left, and the solution of the PDE is to the right*

To see how we can work with more complicated boundary conditions, consider again the heat equation, this time without a source term $u_{xx}+u_{yy}=0$, but with the following boundary conditions: $u(x=0)=3$, $u(x=1)=-1$, $u(y=0)=-5$, and $u(y=1)=5$. This problem was solved with the FDM method earlier in this chapter. Let's solve this problem again using FEM. As in the previous example, let's define a mesh for the problem domain, the function space, and trial and test function objects.

```
In [70]: V = dolfin.FunctionSpace(mesh, 'Lagrange', 1)
In [71]: u = dolfin.TrialFunction(V)
In [72]: v = dolfin.TestFunction(V)
```

Next, define the weak form of the PDE. Here, set *f* = 0 using a `dolfin.Constant` object to represent *f*.

```
In [73]: a = dolfin.inner(dolfin.nabla_grad(u), dolfin.nabla_grad(v)) * dolfin.dx
In [74]: f = dolfin.Constant(0.0)
In [75]: L = f * v * dolfin.dx
```

It remains to define the boundary conditions according to the given specification. In this example, we do not want a uniform boundary condition that applies to the entire boundary, so we need to use the first argument to the boundary selection function passed to the `DirichletBC` class to single out different parts of the boundary. To this end, define four functions that select the top, bottom, left, and right boundaries.

```
In [76]: def u0_top_boundary(x, on_boundary):
    ...:     # on boundary and y == 1 -> top boundary
    ...:     return on_boundary and abs(x[1]-1) < 1e-5
In [77]: def u0_bottom_boundary(x, on_boundary):
    ...:     # on boundary and y == 0 -> bottom boundary
    ...:     return on_boundary and abs(x[1]) < 1e-5
In [78]: def u0_left_boundary(x, on_boundary):
    ...:     # on boundary and x == 0 -> left boundary
    ...:     return on_boundary and abs(x[0]) < 1e-5
In [79]: def u0_right_boundary(x, on_boundary):
    ...:     # on boundary and x == 1 -> left boundary
    ...:     return on_boundary and abs(x[0]-1) < 1e-5
```

The values of the unknown function at each of the boundaries are simple constants that we can represent with instances of `dolfin.Constant`. Thus, we can create instances of `DirichletBC` for each boundary, and the resulting objects are collected in a list `bcs`.

```
In [80]: bc_t = dolfin.DirichletBC(V, dolfin.Constant(5), u0_top_boundary)
    ...: bc_b = dolfin.DirichletBC(V, dolfin.Constant(-5), u0_bottom_boundary)
    ...: bc_l = dolfin.DirichletBC(V, dolfin.Constant(3), u0_left_boundary)
    ...: bc_r = dolfin.DirichletBC(V, dolfin.Constant(-1), u0_right_boundary)
In [81]: bcs = [bc_t, bc_b, bc_r, bc_l]
```

With this specification of the boundary conditions, we can continue to solve the PDE problem by calling `dolfin.solve`. The resulting vector converted to a NumPy array is used for plotting the solution using Matplotlib's pcolor function. The result is shown in Figure 11-8. By comparing the result from the corresponding FDM computation, shown in Figure 11-2, we can conclude that the two methods give the same results.

```
In [82]: u_sol = dolfin.Function(V)
In [83]: dolfin.solve(a == L, u_sol, bcs)
In [84]: u_mat = np.array(u_sol.vector()).reshape(N1+1, N2+1)
In [85]: x = np.linspace(0, 1, N1+2)
    ...: y = np.linspace(0, 1, N1+2)
    ...: X, Y = np.meshgrid(x, y)
In [86]: fig, ax = plt.subplots(1, 1, figsize=(8, 6))
    ...: c = ax.pcolor(X, Y, u_mat, vmin=-5, vmax=5, cmap=mpl.cm.get_cmap('RdBu_r'))
    ...: cb = plt.colorbar(c, ax=ax)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: cb.set_label(r"$u(x_1, x_2)$", fontsize=18)
```

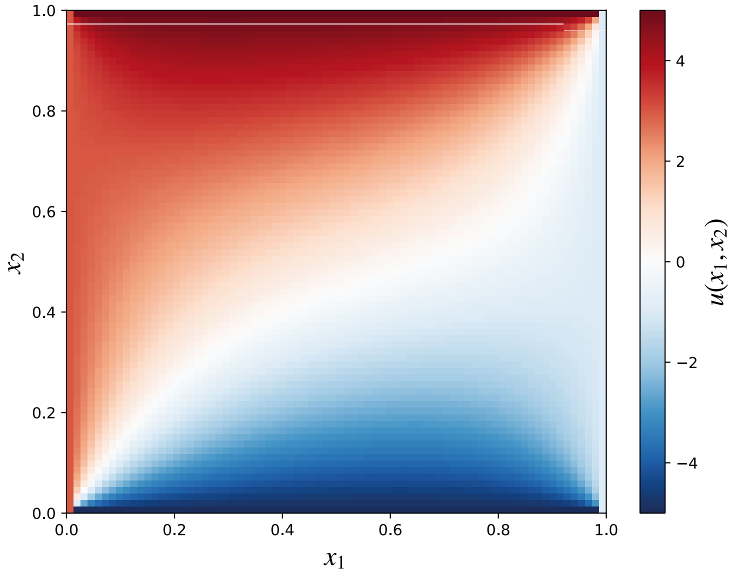

Figure 11-8. *The steady-state solution to the heat equation with different Dirichlet boundary condition on each of the sides of the unit square*

So far, we have used FEM to solve the same kind of problems that we also solved with FDM, but the true strength of FEM becomes apparent first when a PDE problem with more complicated problem geometries is considered. As an illustration, consider the heat equation on a unit circle perforated by five smaller circles, one centered at the origin and the other four smaller circles, as shown in Figure 11-9. We can use the mshr library distributed with FEniCS to generate meshes for geometries like this one. It provides geometric primitives (Point, Circle, Rectangle, etc.) that can be used in algebraic (set) operations to compose a mesh for the problem domain of interest. First, create a unit circle centered at (0, 0) using mshr.Circle, and subtract other Circle objects corresponding to the part of the mesh that should be removed. The resulting mesh is shown in Figure 11-9.

```
In [87]: r_outer = 1
    ...: r_inner = 0.25
    ...: r_middle = 0.1
    ...: x0, y0 = 0.4, 0.4
In [88]: segments = 100
    ...: domain = mshr.Circle(dolfin.Point(.0, .0), r_outer, segments) \
    ...:     - mshr.Circle(dolfin.Point(.0, .0), r_inner, segments) \
    ...:     - mshr.Circle(dolfin.Point( x0,  y0), r_middle, segments) \
    ...:     - mshr.Circle(dolfin.Point( x0, -y0), r_middl, segments) \
    ...:     - mshr.Circle(dolfin.Point(-x0,  y0), r_middle, segments) \
    ...:     - mshr.Circle(dolfin.Point(-x0, -y0), r_middle, segments)
In [89]: mesh = mshr.generate_mesh(domain, 10)
```

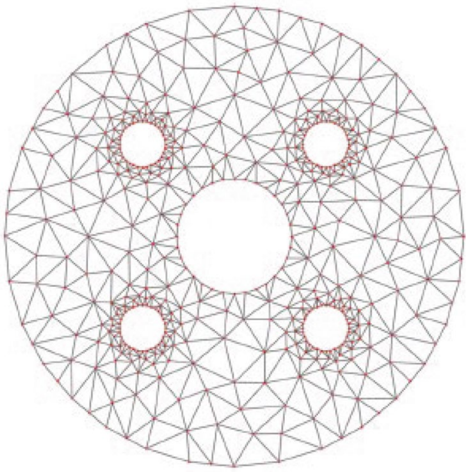

Figure 11-9. *A mesh object generated by the mshr library*

A physical interpretation of this mesh is that the geometry is a cross-section of five pipes through a block of material, where, for example, the inner pipe carries a hot fluid and the middle pipes a cold fluid for cooling the material block (e.g., an engine cylinder surrounded by cooling pipes). With this interpretation in mind, set the boundary condition of the inner pipe to a high value, $u_0(x,y)\big|_{x^2+y^2=r_{\text{inner}}^2}=10$, and the smaller surrounding pipes to a lower value, $u_0(x,y)\big|_{(x-x_0)^2+(y-y_0)^2=r_{\text{middle}}^2}=0$, where (x_0, y_0) is the center of each smaller pipe. Leave the outer boundary unspecified, which is equivalent to the special case of a Neumann boundary condition: $\frac{\partial u(x)}{\partial n}=0$. As before, define functions for singling out vertices on the boundary. Since there are different boundary conditions, we must also use the coordinate argument x to determine which vertices belong to which boundary.

```
In [90]: def u0_inner_boundary(x, on_boundary):
    ...:     x, y = x[0], x[1]
    ...:     return on_boundary and abs(np.sqrt(x**2 + y**2) - r_inner) < 5e-2
In [91]: def u0_middle_boundary(x, on_boundary):
    ...:     x, y = x[0], x[1]
    ...:     if on_boundary:
    ...:         for _x0 in [-x0, x0]:
    ...:             for _y0 in [-y0, y0]:
    ...:                 if abs(np.sqrt((x-_x0)**2+(y-_y0)**2) - r_middle) < 5e-2:
    ...:                     return True
    ...:     return False
In [92]: bc_inner = dolfin.DirichletBC(V, dolfin.Constant(10), u0_inner_boundary)
    ...: bc_middle = dolfin.DirichletBC(V, dolfin.Constant(0), u0_middle_boundary)
In [93]: bcs = [bc_inner, bc_middle]
```

Once the mesh and boundary conditions are specified, we can proceed as usual with defining the function space and the trial and test functions and constructing the weak form representation of the PDE problem.

```
In [94]: V = dolfin.FunctionSpace(mesh, 'Lagrange', 1)
In [95]: u = dolfin.TrialFunction(V)
In [96]: v = dolfin.TestFunction(V)
In [97]: a = dolfin.inner(dolfin.nabla_grad(u), dolfin.nabla_grad(v)) * dolfin.dx
In [98]: f = dolfin.Constant(0.0)
In [99]: L = f * v * dolfin.dx
In [100]: u_sol = dolfin.Function(V)
```

Solving and visualizing the problem also follows the same pattern as before. The result of plotting the solution is shown in Figure 11-10.

```
In [101]: dolfin.solve(a == L, u_sol, bcs)
In [102]: coordinates = mesh.coordinates()
     ...: triangles = mesh.cells()
     ...: triangulation = mpl.tri.Triangulation(
     ...:     coordinates[:, 0], coordinates[:, 1], triangles)
In [103]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
     ...: ax1.triplot(triangulation)
     ...: ax1.set_xlabel(r"$x$", fontsize=18)
     ...: ax1.set_ylabel(r"$y$", fontsize=18)
     ...: c = ax2.tripcolor(
     ...:      triangulation, np.array(u_sol.vector()), cmap=mpl.cm.get_cmap("Reds"))
     ...: cb = plt.colorbar(c, ax=ax2)
     ...: ax2.set_xlabel(r"$x$", fontsize=18)
     ...: ax2.set_ylabel(r"$y$", fontsize=18)
     ...: cb.set_label(r"$u(x, y)$", fontsize=18)
     ...: cb.set_ticks([0.0, 5, 10, 15])
```

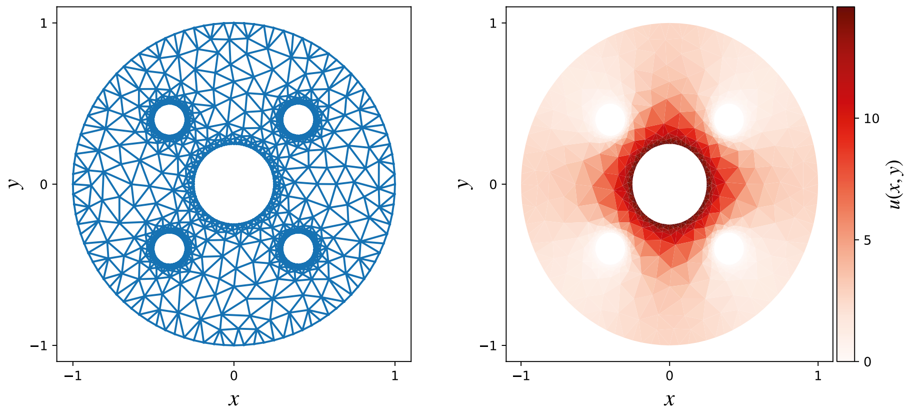

Figure 11-10. *The solution to the heat equation on a perforated unit circle*

Problems with this kind of geometry are difficult to treat with FDM methods but can be handled with relative ease using FEM. Once we obtain a solution for a FEM problem, even for intricate problem boundaries, we can also, with relative ease, post-process the solution function in other ways than plotting it. For example, we might be interested in the value of the function along one of the boundaries. For instance, in the current problem, it is natural to look at the temperature along the outer radius of the problem domain, for example, to see how much the exterior temperature of the body decreases due to the four cooling pipes.

To do this kind of analysis, we need a way of singling out the boundary values from the `u_sol` object. We can do this by defining an object that describes the boundary (here using `dolfin.AutoSubDomain`) and applying it to a new `Function` object used as a mask for selecting the desired elements from the `u_sol` and `mesh.coordinates()`. The following calls this mask function `mask_outer`.

```
In [104]: outer_boundary = dolfin.AutoSubDomain(
     ...:     lambda x, on_bnd: on_bnd and
     ...:         abs(np.sqrt(x[0]**2 + x[1]**2) - r_outer) < 5e-2)
In [105]: bc_outer = dolfin.DirichletBC(V, 1, outer_boundary)
In [106]: mask_outer = dolfin.Function(V)
In [107]: bc_outer.apply(mask_outer.vector())
In [108]: u_outer = u_sol.vector()[mask_outer.vector() == 1]
In [109]: x_outer = mesh.coordinates()[mask_outer.vector() == 1]
```

These steps created the mask for the outer boundary condition and applied it to `u_sol.vector()` and `mesh.coordinates()`, thereby obtaining the function values and the coordinates for the outer boundary points. Next, plot the boundary data as a function of the angle between the (x, y) point and the x axis. The result is shown in Figure 11-11.

```
In [110]: phi = np.angle(x_outer[:, 0] + 1j * x_outer[:, 1])
In [111]: order = np.argsort(phi)
In [112]: fig, ax = plt.subplots(1, 1, figsize=(8, 4))
     ...: ax.plot(phi[order], u_outer[order], 's-', lw=2)
     ...: ax.set_ylabel(r"$u(x,y)$ at $x^2+y^2=1$", fontsize=18)
     ...: ax.set_xlabel(r"$\phi$", fontsize=18)
     ...: ax.set_xlim(-np.pi, np.pi)
```

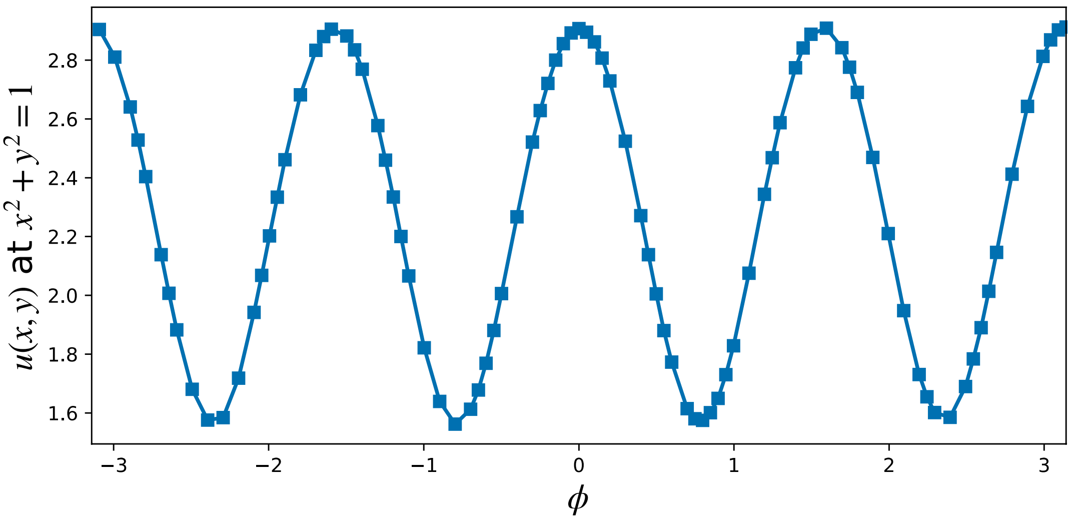

Figure 11-11. *Temperature distribution along the outer boundary of the perforated unit circle*

The accuracy of the solution to a PDE computed with FEM is intimately connected to the element sizes in the mesh that represent the problem domain: a finer mesh gives a more accurate solution. However, increasing the number of elements in the mesh also makes the problem more computationally demanding to solve. Thus, there is a trade-off between the accuracy of the mesh and the available computational resources that must be considered. A mesh with nonuniformly distributed elements is an important tool for dealing with this trade-off. With such a mesh, we can use smaller elements where the unknown function is expected to change in value quickly and fewer elements in less interesting regions. The `dolfin` library provides a simple way to refine a mesh using the `dolfin.refine` function. It takes a mesh as the first argument, and if no other arguments are given, it uniformly refines the mesh and returns a new mesh. However, the `dolfin.refine` function also accepts an optional second argument describing which mesh parts should be refined. This argument should be an instance of a Boolean-valued `dolfin.MeshFunction`, which acts as a mask that flags which elements (cells) should be divided. For example, consider a mesh for the unit circle less the part in the quadrant where $x > 0$ and $y < 0$. We can construct a mesh for this geometry using `mshr.Circle` and `mshr.Rectangle`.

```
In [113]: domain = mshr.Circle(dolfin.Point(.0, .0), 1.0) \
     ...:     - mshr.Rectangle(dolfin.Point(0.0, -1.0), dolfin.Point(1.0, 0.0))
In [114]: mesh = mshr.generate_mesh(domain, 10)
```

The resulting mesh is shown in the left part of Figure 11-12. Using meshes with finer structures near sharp corners in the geometry is often desirable. For this example, it is reasonable to attempt to refine the mesh around the edge near the origin. To do this, we need to create an instance of `dolfin.MeshFunction`; initialize all its elements to `False`, using the `set_all` method; iterate through the elements and mark those near the origin as `True`; and finally call the `dolfin.refine` function with the mesh and the `MeshFunction` instance as arguments. We can do this repeatedly until a sufficiently fine mesh is obtained. The following iteratively calls `dolfin.refine` with a decreasing number of cells marked for splitting.

```
In [115]: refined_mesh = mesh
     ...: for r in [0.5, 0.25]:
     ...:      cell_markers = dolfin.MeshFunction("bool", refined_mesh, dim=2)
     ...:     cell_markers.set_all(False)
     ...:     for cell in dolfin.cells(refined_mesh):
     ...:         if cell.distance(dolfin.Point(.0, .0)) < r:
     ...:              # mark cells within a radius r from the origin to be split
     ...:             cell_markers[cell] = True
     ...:     refined_mesh = dolfin.refine(refined_mesh, cell_markers)
```

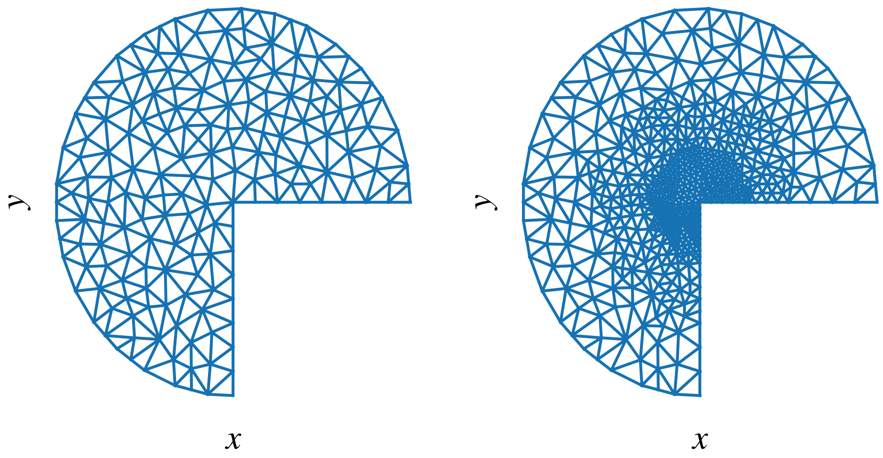

Figure 11-12. *The original and the refined meshes for three-quarters of the unit circle*

The resulting mesh `refined_mesh` is a version of the original mesh with finer element partitioning near the origin. The following code plots the two meshes for comparison, and the result is shown in Figure 11-12.

```
In [116]: def mesh_triangulation(mesh):
     ...:     coordinates = mesh.coordinates()
     ...:     triangles = mesh.cells()
     ...:     triangulation = mpl.tri.Triangulation(
     ...:         coordinates[:, 0], coordinates[:, 1],
     ...:     return triangulation
In [117]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
     ...:
     ...: ax1.triplot(mesh_triangulation(mesh))
     ...: ax2.triplot(mesh_triangulation(refined_mesh))
     ...:
     ...: # hide axes and ticks
     ...: for ax in [ax1, ax2]:
     ...:     for side in ['bottom','right','top','left']:
     ...:         ax.spines[side].set_visible(False)
     ...:         ax.set_xticks([])
     ...:         ax.set_yticks([])
     ...:         ax.xaxis.set_ticks_position('none')
     ...:         ax.yaxis.set_ticks_position('none')
     ...:
     ...: ax.set_xlabel(r"$x$", fontsize=18)
     ...: ax.set_ylabel(r"$y$", fontsize=18)
```

Refining a mesh using `dolfin.refine` is a practical technique for improving simple meshes that are constructed using expressions of geometrical primitives, like the one used in this chapter. As a final example of using FEniCS, let's consider another example of the steady-state heat equation, using this refined mesh for the three-quarters of the unit circle, where we impose Neumann boundary conditions on the vertical and horizontal boundaries along the missing quarter of the unit circle: for the vertical edge, assume an outflux of heat described by $\nabla u \cdot \boldsymbol{n} = -2, x = 0, y < 0$, and through the horizontal edge, assume an influx of heat described by $\nabla u \cdot \boldsymbol{n} = 1, x > 0, y = 0$, while the outer radial boundary is assumed to be described by the Dirichlet boundary condition $u(x, y) = 0, x^2 + y^2 = 1$.

Let's begin, as usual, by creating objects for the function space, the test function, and the trial function.

```
In [118]: mesh = refined_mesh
In [119]: V = dolfin.FunctionSpace(mesh, 'Lagrange', 1)
In [120]: u = dolfin.TrialFunction(V)
In [121]: v = dolfin.TestFunction(V)
```

For problems with Neumann boundary conditions, we need to include the boundary condition in the weak form of the PDE. Recall that the weak form for the Poisson equation is $\int_\Omega \nabla u.\nabla v \ \mathrm{d}x = \int_\Omega fv \ \mathrm{d}x + \int_{\Gamma_N} g \ v \ d\Gamma$, so compared to the earlier examples, we need to account for the additional term $\int_{\Gamma_N} g \ v \ d\Gamma$, which is an integral over the boundary with Neumann boundary condition. To represent the integral measure for this integral in the weak form specification, we can use `dolfin.ds`, but to be able to distinguish different parts of the boundary, we first need to mark the boundary parts. One way to do this in FEniCS is to use a `dolfin.MeshFunction` object and assign to it a unique integer value for each distinct part of the boundary. First, create a `dolfin.MeshFunction` instance.

```
In [122]: boundary_parts = dolfin.MeshFunction(
     ...:     "size_t", mesh, mesh.topology().dim()-1)
```

Next, define a function for selecting boundary points and a `dolfin.AutoSubDomain` instance that is initialized from the boundary selection function. The `AutoSubDomain` instance can then mark the corresponding cells in the `MeshFunction` (here called `boundary_parts`) with an identifying integer value. The following lines of code perform these steps for the vertical edge of the mesh, where $x = 0$ and $y < 0$.

```
In [121]: def v_boundary_func(x, on_boundary):
     ...:     """ the vertical edge of the mesh, where x = 0 and y < 0 """
     ...:     x, y = x[0], x[1]
     ...:     return on_boundary and abs(x) < 1e-4 and y < 0.0
In [122]: v_boundary = dolfin.AutoSubDomain(v_boundary_func)
In [123]: v_boundary.mark(boundary_parts, 0)
```

Repeat the same procedure for the horizontal edge of the mesh, where $y = 0$ and $x > 0$.

```
In [124]: def h_boundary_func(x, on_boundary):
     ...:     """ the horizontal edge of the mesh, where y = 0 and x > 0 """
     ...:     x, y = x[0], x[1]
     ...:     return on_boundary and abs(y) < 1e-4 and x > 0.0
In [125]: h_boundary = dolfin.AutoSubDomain(h_boundary_func)
In [126]: h_boundary.mark(boundary_parts, 1)
```

We can also use the same method to define Dirichlet boundary conditions. Mark the part of the boundary described by the Dirichlet boundary condition and then use it to create the `dolfin.DirichletBC` object.

```
In [127]: def outer_boundary_func(x, on_boundary):
     ...:     x, y = x[0], x[1]
     ...:     return on_boundary and abs(x**2 + y**2-1) < 1e-2
In [128]: outer_boundary = dolfin.AutoSubDomain(outer_boundary_func)
In [129]: outer_boundary.mark(boundary_parts, 2)
In [130]: bc = dolfin.DirichletBC(V, dolfin.Constant(0.0), boundary_parts, 2)
```

Once the boundaries are marked, we can create the weak form of the PDE. Since a partitioned boundary is used here, we must specify the `domain` and `subdomain` arguments to the integral measures `dolfin.dx` and `dolfin.ds`, using the `mesh` and `boundary_parts` objects.

```
In [131]: dx = dolfin.dx(domain=mesh, subdomain_data=boundary_parts)
In [132]: a = dolfin.inner(dolfin.nabla_grad(u), dolfin.nabla_grad(v)) * dx
In [133]: f = dolfin.Constant(0.0)
In [134]: g_v = dolfin.Constant(-2.0)
In [135]: g_h = dolfin.Constant(1.0)
In [136]: L = f * v * dolfin.dx(domain=mesh, subdomain_data=boundary_parts)
In [137]: L += g_v * v * dolfin.ds(0, domain=mesh, subdomain_data=boundary_parts)
In [138]: L += g_h * v * dolfin.ds(1, domain=mesh, subdomain_data=boundary_parts)
```

The last two code cells added new terms for the Neumann boundary conditions for the mesh's vertical and horizontal edges. These parts of the boundary are denoted by integers 0 and 1, respectively, as defined in the preceding section, and these integers are passed as an argument to the `dolfin.ds` to select integration over different parts of the boundaries.

```
In [139]: u_sol = dolfin.Function(V)
In [140]: dolfin.solve(a == L, u_sol, bc)
```

Once the representation of the weak form of the PDE is defined, we can go ahead and solve the problem using `dolfin.solve`, as done in earlier examples. Finally, plot the solution using Matplotlib's triangulation plot functions. The results are shown in Figure 11-13. The graph shows that, as expected, the solution has more structure near the edge at the origin. Using a mesh with smaller elements in this region is a good way to obtain sufficient resolution without inflicting excessive computational cost by using a uniformly fine-structured mesh.

```
In [141]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
     ...: triangulation = mesh_triangulation(mesh)
     ...: ax1.triplot(triangulation)
     ...: ax1.set_xlabel(r"$x$", fontsize=18)
     ...: ax1.set_ylabel(r"$y$", fontsize=18)
     ...:
     ...: data = np.array(u_sol.vector())
     ...: norm = mpl.colors.Normalize(-abs(data).max(), abs(data).max())
     ...: c = ax2.tripcolor(triangulation, data, norm=norm,
     ...:     cmap=mpl.cm.get_cmap("RdBu_r"))
     ...: cb = plt.colorbar(c, ax=ax2)
     ...: ax2.set_xlabel(r"$x$", fontsize=18)
     ...: ax2.set_ylabel(r"$y$", fontsize=18)
     ...: cb.set_label(r"$u(x, y)$", fontsize=18)
     ...: cb.set_ticks([-.5, 0, .5])
```

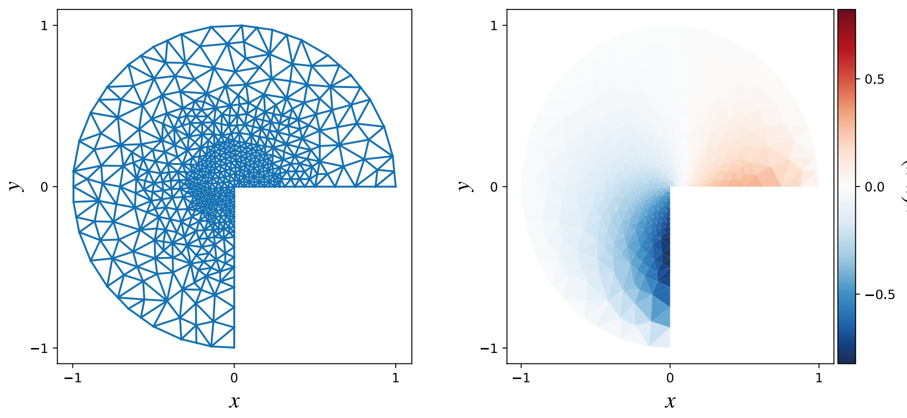

Figure 11-13. *Solution to the heat equation on a quarter of the unit circle with Neumann and Dirichlet boundary conditions*

The examples explored in this section are merely a few simple demonstrations of the types of problems for which the FEniCS framework can be used. There are many features in FEniCS that I have not even been able to mention here. For the reader who is particularly interested in solving PDE problems, I recommend studying the `FEniCS book (Anders Logg, 2012)` and its many example applications. In particular, important aspects of solving PDEs with FEM that I have not been able to discuss here are nontrivial Neumann boundary conditions (which need to be included in the formulation of the weak form of the PDE), PDEs for vector-valued functions, higher-dimensional PDE problems (e.g., the heat equation in three dimensions), and time-dependent PDE problems. These topics, and many others, are well supported in the FEniCS framework.

Summary

This chapter examined methods for solving partial differential equations (PDEs) and how these methods can be employed within the scientific Python environment. Specifically, it introduced the finite-difference method (FDM) and the finite-element method (FEM) for solving PDE problems and used these methods to solve several example problems. The advantage of FDM is its simplicity, and it is a very practical method for problems where it is easily applicable (simple problem domains, uniform discretization, etc.). For more complicated PDE problems, for example, where the problem domain is more complex, FEM is generally more suitable. However, the mathematical theory of the FEM is more involved, and the implementation is far more technical. While several advanced FEM frameworks can be used in Python, this chapter focused on one prominent example: the FEniCS framework. FEniCS is a full-featured FEM software that can be used for a wide range of PDE problems. With the examples considered here, we have only scraped the surface of what can be achieved with the software. However, the hope is that the examples studied in this chapter give a general sense of the workflow when solving PDE problems with FEM and when using the FEniCS software, in particular.

Further Reading

Although this chapter discussed FDM and FEM, there are other successful and useful methods for numerically solving PDEs. For instance, the finite-volume method (FVM) is a variant of the FEM method that is often used for fluid dynamics calculations, as well as in other fields. The Python library FiPy provides a framework for solving PDE problems using this method, and a theoretical introduction to the method is given in *Principles of Computational Fluid Dynamics* by P. Wesseling (Springer, 2009). The theoretical background information about the FDM and FEM that is given in this chapter is very brief indeed, and it merely serves to introduce the terminology and notation used here. For serious work with the FDM, and particularly the FEM method, it is important to understand the fundamentals of these methods thoroughly.

Good introductions to FDM and FEM are given in M. Gockenbach's *Partial Differential Equations* (SIAM, 2011) and *Understanding and Implementing the Finite Element Method* (SIAM, 2006), *Numerical Solution of Partial Differential Equations by the Finite Element Method* by C. Johnson (Dover, 2009), and *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems* by R. LeVeque (SIAM, 2007).

The FEniCS book, *Automated Solution of Differential Equations by the Finite Element Method* by A. Logg et al., (Springer, 2012), is available for free online from the FEniCS project's website (`http://fenicsproject.org`), also contains a nice introduction to the FEM method, in addition to detailed documentation of the FEniCS software itself.